Hop, Skip, Jump

Implementing a concurrent interpreter with Promises

Timothy Jones

Victoria University of Wellington

`tim@ecs.vuw.ac.nz`

January 14, 2015

This Talk

Implementing a fully asynchronous JavaScript application

Using Promises to tame asynchronous behaviour

Extending the standard Promises with custom behaviour

The costs of naïve implementation

Grace

A programming language

Grace

```
object {
  var name := "Bob"
  method talk {
    print "My name is {name}"
  }
}

if (x > 1) then {
  while { x < y } do {
    x := x ˆ 2
  }
}
```

Implementation

Minigrace, compiling to C and JavaScript

Trial courses running in simple Web-based editor

Runtime execution in the browser

Browser Execution

The editing environment occupies the same runtime as the code

How might we implement the while-do loop as a JavaScript function?

```
function whileDo(condBlock, doBlock) {
  while (condBlock.apply()) {
    doBlock.apply();
  }
}
```

Browser Execution

Whoops!

```
while { true } do {}
```

Cannot use threads (without losing direct access to the DOM)

The Hop

Hopper is a Grace interpreter written in asynchronous JavaScript

Asynchronous JavaScript is awful to write

- Pyramid of Doom
- Difficult control flow (no guarantee of code ordering)
- Inherently non-composable
- Explicit error handling everywhere

Async JS

```
var current = 0, total = urls.length;
for (var i = 0; i < total; i++) {
  get(url, function (err) {
    if (err) console.error("failed to pull data");
    if (++current === total)
      console.log(urls[i] + " was last");
  });
  console.log("reading " + url);
}
```

Async JS

Hopper started out this way

Quickly filled complexity budget

Promises

Promises, or Futures, are a standard solution to this problem

Encode the concept of an asynchronous operation as a value

All interactions are asynchronous

- Detecting if the operation has finished
- Retrieving the result of the operation
- Performing a subsequent operation

Then

All of those interactions are the same thing!

```
get(url1).then(function (contents) {
  post(url2, contents);
});
```

Then

And now asynchronous actions are composable

```
get(url1).then(function (contents) {
  return post(url2, contents);
});
```

Promises/A+

Promises for JavaScript strictly specified by Promises/A+

Really defines the behaviour of `then`

- A promise is just any object with a conformant `then`

Implementations provide constructors and helper methods

Defining then

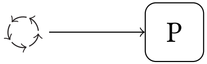
```
promise.then(onFulfilled, onRejected)
```

Both arguments optional, called as appropriate, at most once

They *must not be called* until the stack is empty

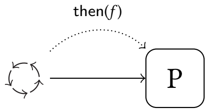
Defining then

Returns a task that represents both executions



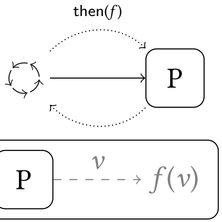
Defining then

Returns a task that represents both executions



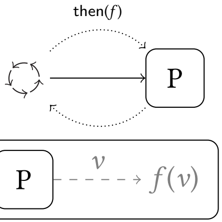
Defining then

Returns a task that represents both executions

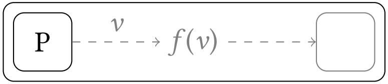


Defining then

Returns a task that represents both executions



If the subsequent function returns a task, that is also included



Pleasant Async

```
function whileDo(condBlock, doBlock) {
  return condBlock.apply().then(function (cond) {
    if (cond) {
      return doBlock.apply().then(function () {
        return whileDo(condBlock, doBlock);
      });
    }
  });
}
```

Tasks

Hopper promises aren't compliant, and so are called Tasks

Can be given a `this` value, which carries on to `then` calls

Why does the stack need to be cleared?

- Effectively equivalent to tail-call optimisation
- JavaScript has a tiny maximum stack height
- Also necessary to preserve non-synchronous nature
- Now efficiently performed with `asap`

Tasks

Tasks can be manually constructed

```
new Task(function (resolve, reject) {
  get(url, function (err, contents) {
    if (err) reject(err);
    else resolve(contents);
  })
});
```

They manually yield to the event loop every 50ms

- Switch to setImmediate instead of asap

Async Methods

Now Grace methods can block execution without blocking the thread

```
var contents := get(url1)
post(url2, contents)
print "Posted to the url"
```

Async Methods

It's also really easy to build lightweight threading

```
function spawn(block) {
  block.apply(); // Yields, will continue in the future
  return new Task(function (resolve) {
    resolve();
  });
}
```

Async Methods

Once the function is exposed to Grace:

```
spawn {
  while { true } do {
    print "spawned"
  }
}

while { true } do {
  print "original"
}
```

Viral Async

We don't know if a method is going to be async

To get a reliable interface, we have to assume *every* method is

What about methods that *must* run synchronously?

- Important for FFI: a Grace object masquerading as a JS object

```
var myTalk := object {
  method speakingTime is synchronous {
    return random.numberFrom(32) to(57)
  }
}
```

Now and Then

The `now` method behaves like `then`, but it must occur synchronously

- If a task is waiting to `asap`, it has a `deferred` method
- This method can be invoked early to force it to run immediately
- If it ends up depending on another task, it also forces that task

If a task is forced (or has nothing to force) but is still not complete, the task resulting from the call to `now` is immediately rejected

- This rejection is visible in Grace

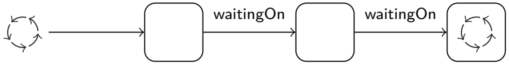
`now` completely breaks the concept of a promise as a black box

Stop

We want to be able to stop running code from the editor

- Would also like this to be modular

Hop from one task to the next, causing the final task to be rejected



stop also breaks the black box

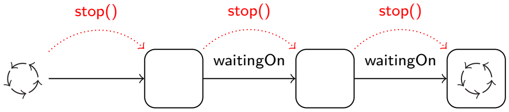
It’s also probably a bad idea: better to kill the interpreter

Stop

We want to be able to stop running code from the editor

- Would also like this to be modular

Hop from one task to the next, causing the final task to be rejected



stop also breaks the black box

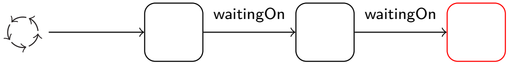
It’s also probably a bad idea: better to kill the interpreter

Stop

We want to be able to stop running code from the editor

- Would also like this to be modular

Hop from one task to the next, causing the final task to be rejected



stop also breaks the black box

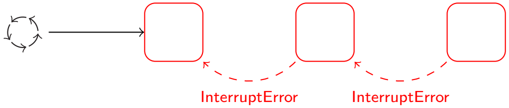
It's also probably a bad idea: better to kill the interpreter

Stop

We want to be able to stop running code from the editor

- Would also like this to be modular

Hop from one task to the next, causing the final task to be rejected



stop also breaks the black box

It's also probably a bad idea: better to kill the interpreter

The Story So Far

Tasks provide consistency in an unpredictable asynchronous world

Black-box approach is incompatible with more complex requirements

- Once you go async, you can’t go back

Hopper uses tasks everywhere! (Even in the parser)

Tasks are Expensive

Yielding to the event loop is an expensive operation

The overall cost of the task machinery is enormous

What can we do to cut down on memory and performance losses?

The Skip

Garbage is the main problem

- Lots of allocations
- Can we take advantage of generational GC?

Analysis

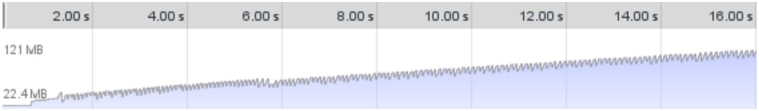
Returning to our original problem

```
while { true } do {}
```

This now no longer hangs the browser

But at what cost?

Analysis



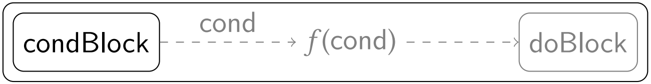
While loops don't run in constant memory!

- *Some* allocation is to be expected
- But nothing is being thrown away here

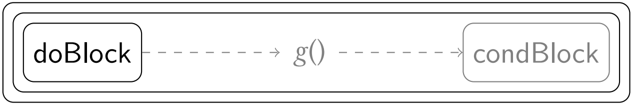
Pleasant Async?

```
function whileDo(condBlock, doBlock) {
  return condBlock.apply().then(function (cond) {
    if (cond) {
      return doBlock.apply().then(function () {
        return whileDo(condBlock, doBlock);
      });
    }
  });
}
```

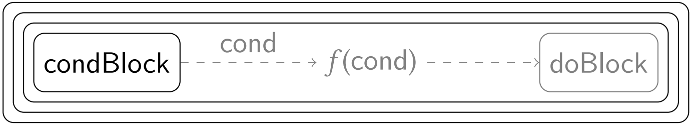
Pleasant Async?



Pleasant Async?



Pleasant Async?



Closure Capture

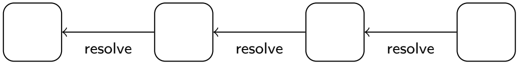
When then is called, it creates a new task

If a function passed to then returns a task, the two are bound together:

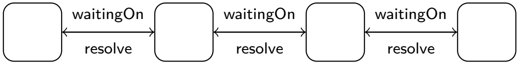
```
new Task(function (resolve) {
  onReady(function (value) {
    if (value instanceof Task && value.isPending) {
      value.then(resolve);
    }
  });
});
```

Each new inner task captures the outer one, creating an implicit chain

Closure Capture



Closure Capture



Breaking the Chain

The capture seems like a necessary part of the behaviour

Weak pointers?

- WeakSet and friends haven't rolled out to many browsers yet
- Hopper should be able to support older browsers

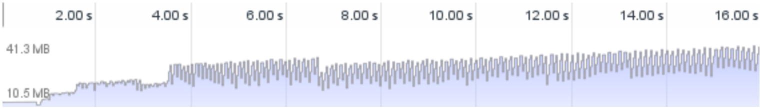
The Simplest Solution

Drop the **returns**

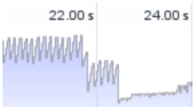
```
function whileDo(condBlock, doBlock) {
  return new Task(function (resolve) {
    (function loop() {
      condBlock.apply().then(function (cond) {
        if (cond) doBlock.apply().then(function () {
          loop();
        });
        else resolve();
}); }()); }); }
```

Loops are better than recursion again

The Simplest Solution



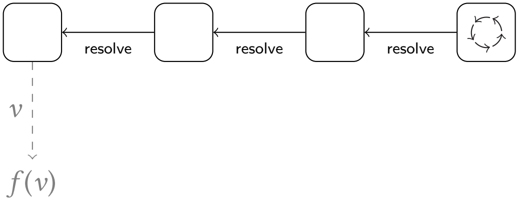
GC is lazy



Task Folding

Idea: Most tasks are just there to pass a value back to another task

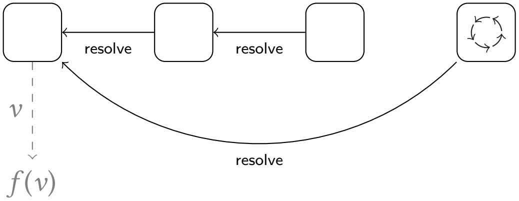
What if we could skip over them?



Task Folding

Idea: Most tasks are just there to pass a value back to another task

What if we could skip over them?



Task Folding

Counterpoint: We can't know whether these tasks have been stored

```
var store;
promise.then(function (value) {
  return store = asyncOperation(value);
}).then(function () {
  console.log("Got to here");
  store.then(function () { console.log("But not here"); });
});
```

Task Folding

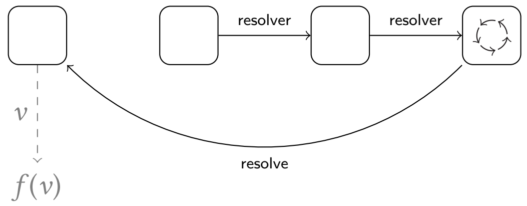
Tasks are reactive!

- The only way to tell if a task is done is by calling `then`
- These tasks are identified by having no other pending callbacks

These tasks can be ‘put to sleep’, and `then` wakes them back up

- They would need to explicitly store the tasks in front
- But this doesn’t prevent them from being GCed

Task Folding



Taking Out The Garbage

Promises are nice but expensive in large (huge) numbers

- It’s not entirely clear if this is avoidable here

It’s okay to allocate a lot, as long you collect a lot

- Optimise the implementation without compromising behaviour

Distant Returns

A **return** always refers to the nearest enclosing method

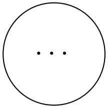
```
method capAtTen(x) {
  if (x > 10) then {
    return 10
  }

  return x
}
```

The Jump

The execution jumps back to the method’s call point

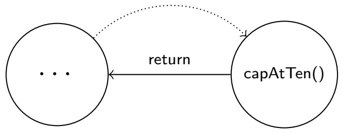
Each return function is a continuation for the call point



The Jump

The execution jumps back to the method's call point

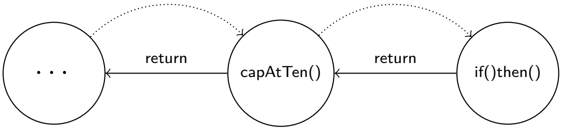
Each return function is a continuation for the call point



The Jump

The execution jumps back to the method's call point

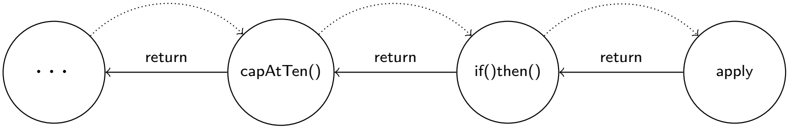
Each return function is a continuation for the call point



The Jump

The execution jumps back to the method's call point

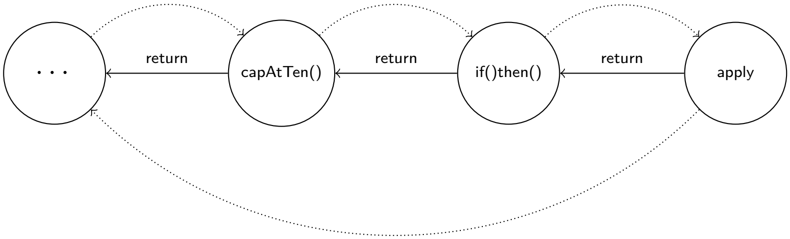
Each return function is a continuation for the call point



The Jump

The execution jumps back to the method's call point

Each return function is a continuation for the call point



The Jump

Each method call has a ‘return’ function

A return calls this function, and terminates its own execution

- (By returning a task that is never resolved or rejected)

Efficient! No need to manually roll back the stack

- The ‘stack’ is a implicitly linked list of functions

Finally

But what if we have to clean up?

```
method broken {
  try { return } finally { mustRun }
}
```

We specifically jump over the finally, completely forgetting about it

- There's no explicit stack, so there's nowhere to remember this
- Rolling back the stack manually wouldn't have this problem

Finally

Solutions?

- Index the callbacks with their position in the stack?
- Can Promise-like objects help us here as well?

Final Points

Promises can still fill the complexity budget

- Large-scale asynchronous behaviour is just hard
- Promises aren't (lightweight) threads!

Like all abstractions, there are costs

- The JavaScript environment is not your friend

$ npm install hopper

`https://github.com/zmthy/hopper`

`https://promisesaplus.com`

`tim@ecs.vuw.ac.nz`